Лабораторная работа № 9 по курсу дискретного анализа: Графы

Выполнил студент группы М8О-308б-22 МАИ *Караев Тариел Жоомартбекович.*

Условие

Задан взвешенный ориентированный граф, состоящий из n вершин и m ребер. Вершины пронумерованы целыми числами от 1 до n. Необходимо найти длины кратчайших путей между всеми парами вершин при помощи алгоритма Джонсона. Длина пути равна сумме весов ребер на этом пути. Обратите внимание, что в данном варианте веса ребер могут быть отрицательными, поскольку алгоритм умеет с ними работать. Граф не содержит петель и кратных ребер.

Метод решения

Так как алгоритм Дейкстры не умеет работать с отрицательными рёбрами, необходимо на время избавиться от них в нашем графе. Для этого мы добавляем в граф фиктивную вершину S и строим из неё рёбра с весом 0 в каждую вершину исходного графа.

Для нового графа запускаем алгоритм Беллмана – Форда, который либо обнаруживает наличие отрицательного цикла в графе и завершает алгоритм, либо возвращает кратчайшие расстояния от фиктивной вершины S до каждой вершины исходного графа. Суть алгоритма заключается в том, что мы $V - 1$ раз проходим по всем рёбрам и релаксируем их, если

$$d[v] > d[u] + w(u, w).$$

Если на V-ой итерации происходит ещё одна релаксация, то в графе имеется отрицательный цикл. С помощью этих кратчайших расстояний мы перевзвешиваем рёбра по следующей формуле:

$$\omega'(u, v) = \omega(u, v) + \varphi(u) - \varphi(v).$$

Удаляем фиктивную вершину и запускаем алгоритм Дейкстры для каждой вершины графа, который возвращает кратчайшие расстояния до каждой другой вершины графа. Для преобразования этих расстояний к изначальному графу необходимо применить обратную формулу перевзвешивания:

$$\omega(u, v) = \omega'(u, v) - \varphi(u) + \varphi(v).$$

Суть алгоритма Дейкстры заключается в том, что в алгоритме поддерживается множество вершин, для которых уже вычислены длины кратчайших путей до них из s. На каждой итерации основного цикла выбирается вершина, не помеченная посещённой, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина добавляется в множество посещённых и производится релаксация всех исходящих из неё рёбер.

Описание программы

Код программы:

```
#include <limits>
#include <optional>
#include <queue>
#include <vector>
#include <cstddef>
#include <iostream>

class Graph {
private:
    struct adjListElem {
        size_t adjNode;
        long long weight;
        adjListElem(size_t adjNode, size_t weight)
            : adjNode(adjNode), weight(weight) {}
        friend bool operator>(const adjListElem &lhs, const adjListElem &rhs)
            return lhs.weight > rhs.weight;
        }
    };

    std::vector<std::vector<adjListElem>> adjList;
    std::vector<std::vector<long long>> adjMatrix;

    std::vector<long long> Dijkstra(size_t u) {
        size_t n = adjList.size();
        std::vector<long long> distances(n,
                                         std::numeric_limits<long long>::max(
        distances[u] = 0;
        std::priority_queue<adjListElem, std::vector<adjListElem>,
                            std::greater<adjListElem>>
            minHeap;
        minHeap.emplace(u, 0);
        std::vector<bool> visited(n, false);
        while (!minHeap.empty()) {
            adjListElem cur = minHeap.top();
            minHeap.pop();
            u = cur.adjNode;
            if (visited[u])
                continue;
            visited[u] = true;
            for (adjListElem el : adjList[u]) {
```

```
                size_t v = el.adjNode;
                long long w = el.weight;
                if (u == v)
                    continue;
                if (distances[u] < distances[v] - w) {
                    distances[v] = distances[u] + w;
                    minHeap.emplace(v, distances[v]);
                }
            }
        }
        return distances;
    }

    std::optional<std::vector<long long>> BellmanFord(size_t source) {
        size_t n = adjList.size();
        std::vector<long long> distances(n,
                                         std::numeric_limits<long long>::max(
        distances[source] = 0;

        for (size_t i = 0; i < n; ++i) {
            for (size_t u = 0; u < n; ++u) {
                for (const adjListElem &edge : adjList[u]) {
                    size_t v = edge.adjNode;
                    long long weight = edge.weight;
                    if (distances[u] < distances[v] - weight) {
                        if (i == n - 1) {
                            return std::nullopt;
                        }
                        distances[v] = distances[u] + weight;
                    }
                }
            }
        }

        return distances;
    }

public:
    Graph(size_t verticesCount) : adjList(verticesCount + 1), adjMatrix(
            verticesCount+1,
            std::vector<long long>(verticesCount+1, std::numeric_limits<long
        for (size_t i = 1; i < verticesCount + 1; ++i) {
```

```
            adjList[0].emplace_back(i, 0);
        }
        for (size_t i = 1; i < verticesCount+1; ++i) {
            adjMatrix[i][i] = 0;
        }
    }

    void AddEdge(size_t u, size_t v, long long weight) {
        adjList[u].emplace_back(v, weight);
        adjMatrix[u][v] = weight;
    }

    std::optional<std::vector<std::vector<long long>>> Johnson() {
        auto potentials = BellmanFord(0);
        if (!potentials) {
            return std::nullopt;
        }
        for (size_t u = 1; u < adjList.size(); ++u) {
            for (auto &el : adjList[u]) {
                auto v = el.adjNode;
                if (el.weight != std::numeric_limits<long long>::max())
                    el.weight = el.weight + (*potentials)[u] - (*potentials)[
            }
        }
        std::vector<std::vector<long long>> res(
            adjList.size(), std::vector<long long>(adjList.size()));
        for (size_t i = 1; i < adjList.size(); ++i) {
            res[i] = Dijkstra(i);
            for (size_t j = 1; j < res[i].size(); ++j) {
                if (res[i][j] != std::numeric_limits<long long>::max())
                    res[i][j] = res[i][j] + (*potentials)[j] - (*potentials)[
            }
        }
        return res;
    }

    std::optional<std::vector<std::vector<long long>>> FloydWarshall() {
        size_t n = adjList.size();
        auto distance = adjMatrix;

        for (size_t k = 1; k < n; ++k) {
            for (size_t i = 1; i < n; ++i) {
```

```cpp
                for (size_t j = 1; j < n; ++j) {
                    if (distance[i][k] != std::numeric_limits<long long>::max
                        distance[k][j] != std::numeric_limits<long long>::max
                        distance[i][j] > distance[i][k] + distance[k][j]) {
                        distance[i][j] = distance[i][k] + distance[k][j];
                    }
                }
            }
        }

        for (size_t i = 1; i < n; ++i) {
            if (distance[i][i] < 0) {
                return std::nullopt;
            }
        }

        return distance;
    }

    Graph() = delete;
};

int main() {
    size_t n, m;
    std::cin >> n >> m;
    Graph graph(n);
    for (size_t i = 0; i < m; ++i) {
        size_t u,v;
        long long w;
        std::cin >> u >> v >> w;
        graph.AddEdge(u, v, w);
    }
    auto allDistances = graph.Johnson();
    if (!allDistances) {
        std::cout << "Negative cycle\n";
        return 0;
    }
    for (size_t i = 1; i < n+1; ++i) {
        for (size_t j = 1; j < n+1; ++j) {
            auto x =(*allDistances)[i][j];
            if (x == std::numeric_limits<long long>::max()) {
                std::cout << "inf ";
```

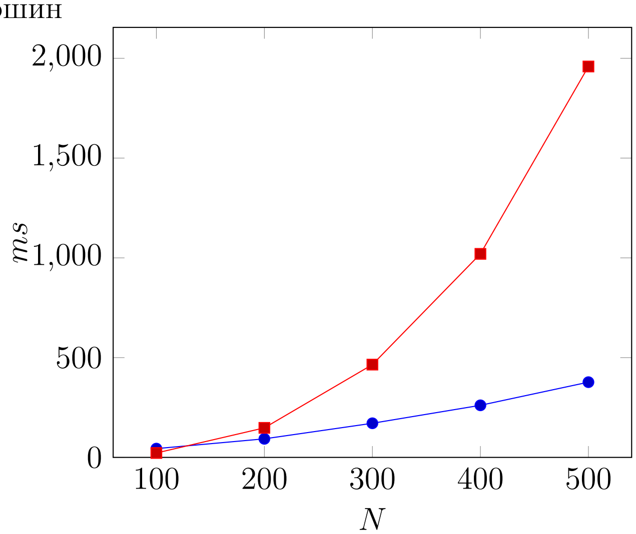
```
            }
            else {
                std::cout << x << ' ';
            }
        }
        std::cout << '\n';
    }
}
```

Дневник отладки

Ошибок не обнаружено.

Тест производительности

Количество ребер будет фиксированным и равным 4000. Будем изменять количество вершин



Красный граф - алгоритм Флойда-Уоршелла, синий - Джонсона. Таким образом видим, что алгоритм Джонсона отлично подходит для разреженного графа. А Флойд-Уоршелл может обгонять его на полных графах, что видно в начале (у графа с 100 вершин максимальное количество ребер - 4950).

Сложность алгоритма Джонсона - $O(n(n+m)logn)$, т.к. мы используем алгоритм Дейкстры со сложностью $O((n+m)logn)$ n раз.

Выводы

В ходе выполнения лабораторной работы я изучил алгоритм Джонсона и применил его для решения задачи нахождения кратчайших путей между всеми парами вершин в графе. Полученные знания позволили успешно реализовать и проверить работу алгоритма на практике.